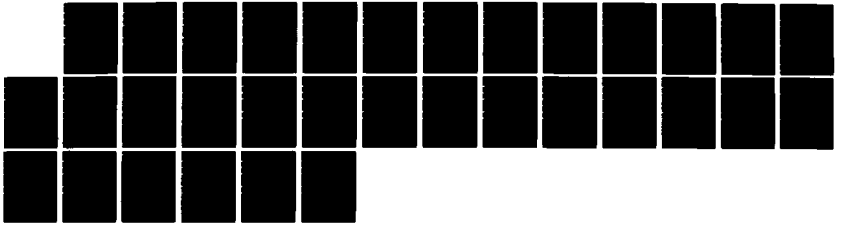
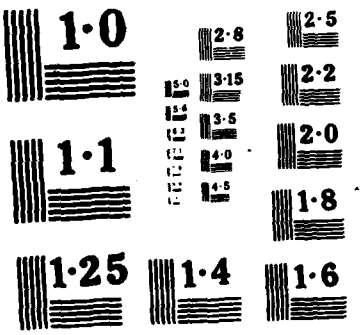
AD-A170 360 A CONNECTIONIST SIMULATOR FOR THE BBN BUTTERFLY MULTIPROCESSOR(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE M FANTY JAN 86 TR-164 N00014-84-K-0655 1/1

UNCLASSIFIED F/G 9/2 NL



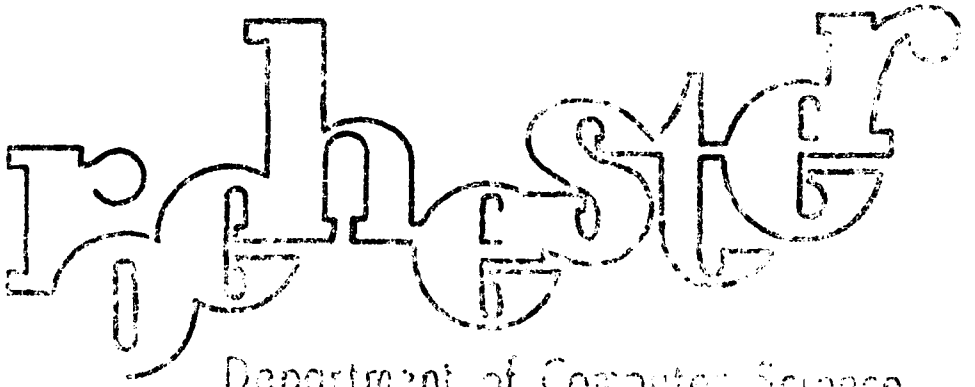

1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6

(12)

A Connectionist Simulator for the BBN Butterfly Multiprocessor

Mark Fanty
Department of Computer Science
University of Rochester
Rochester, NY 14627

TR 164
January, 1986



Department of Computer Science
University of Rochester
Rochester, New York 14627

A Connectionist Simulator for the BBN Butterfly Multiprocessor

Mark Fanty
Department of Computer Science
University of Rochester
Rochester, NY 14627

TR 164
January, 1986

This report describes the implementation of a connectionist simulator on the BBN Butterfly Multiprocessor. It presents the model of connectionist networks used by the simulator and gives a detailed account of the simulator's implementation. Performance data for some sample networks is included. Some plans for future improvements are discussed.

This work was supported in part by the National Science Foundation under grant number DCR-8320136 and in part by the Office of Naval Research under grant number N00014-84-K-0655.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 164	2. GOVT ACCESSION NO. ADA 170360	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Connectionist Simulator for the BBN Butterfly Multiprocessor		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mark Fanty		8. CONTRACT OR GRANT NUMBER(s) N00014-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Rochester Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/1400 Wilson Blvd. Arlington,VA 22209		12. REPORT DATE January 1986 13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Inofrmation Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel simulation
connectionist networks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the implementation of a connectionist simulator on the BBN Butterfly Multiprocessor. It presents the model of connectionist networks used by the simulator and gives a detailed account of the simulator's implementation. Performance data for some sample networks is included. Some Plans for future improvements are discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Introduction

Connectionist networks consist of simple elements which communicate by sending their level of activation over links to other elements. Connectionist units (hereafter just "units") are patterned after neurons, but they are not exact neural models. They have a small number of states and perform simple computations. The behavior of a network is determined by the pattern of connections and the weights on the links. Connectionism has been gaining popularity among some AI researchers and other cognitive scientists (special issue *Cognitive Science*, vol. 9. no. 1, January-March 1985; Rumelhart & McClelland, 1986).

As with other computational models, it is important for the researcher to be able to implement and test. The Rochester Connectionist Simulator (forthcoming) evolved to allow the user to build and execute connectionist networks. It is designed to be as flexible as possible. Each unit can compute a different function of its inputs (functions are provided by the user), and links may be made arbitrarily. This flexibility exacts a cost in efficiency. Each unit and link must be explicitly represented by a data structure. Efficiency is an issue as connectionist networks can easily grow large enough to overwhelm any existing computer. (The human brain presumably computes in the connectionist style and uses billions of neurons with thousands of connections each.)

A version of the simulator was implemented on the BBN Butterfly Multiprocessor in order to increase the size of networks which can be simulated efficiently. The Butterfly at the University of Rochester consists of 120 Motorolla 68000 processors with a Megabyte of memory each. The speedup achieved is quite good, as described in section 5.4.

The primary purpose of this report is to describe the implementation of the connectionist simulator on the Butterfly. To that end, an overview of the connectionist approach is given in section 2. Section 3 describes the simulator from the user's point of view. Section 4 provides a brief overview of the Butterfly and describes the implementation in detail. A basic familiarity with the Butterfly and its operating system, Chrysalis, would facilitate a careful reading of this section. Details of the implementation are given because of the novel architecture and because this report is, in part, intended as a guide to other Butterfly programmers. But this section can also be read at a more general level ignoring the particular Chrysalis calls used. Section 5 presents performance data for some sample networks. Section 6 describes ideas for future improvements.

2. Connectionist Networks

A detailed discussion of connectionist models may be found in Feldman & Ballard (1982). In this section, I will attempt to give enough of an overview to allow the reader to understand the requirements of a simulator. The connectionist model is not firmly fixed. Generally speaking, the simpler the better from a scientific standpoint. Simple models can be more difficult to work with however. The philosophy of the simulator to give the user as much freedom as possible.

A-1

Connectionist networks consist of some number of simple units with links between them (see Figure 1). Each unit has a *potential*, or level of activation. In Feldman & Ballard (1982) this is a real value in [-10,10], though the simulator does not impose this restriction. Each unit also has an integer *output* value in the range [0,10] which is typically the nearest integer to the potential but can include, for example, random perturbations. The output is meant to correspond to the rate of firing in a neural model. This output value is transmitted down all links emanating from that unit. Each unit also has an internal *state*, which can influence its response to inputs. The number of internal states should be fairly small in order to keep the units "simple". The units are updated synchronously. Each computes a new potential, state and output as a function of the inputs and the old potential, state and output.

The links may have *weights*, so the value reaching a unit is a function of the output of the source unit and the weight on the link. Typically, a negative weight means the link is inhibitory; a positive weight means the link is excitatory; a weight of zero is equivalent to no link. The weighting occurs at the receiving unit. Weights may be changed dynamically as a function of the old weight, the potential and state of the receiving unit, and the *history*. The history of a link is a single value which is meant to encode recent activity of the link. It was added for modelling learning algorithms in which weights are increased when the unit is active and the link has been active recently. At each step it is updated as a function of the current input and the old history.

In addition, units may have more than one *site* at which to receive inputs. This allows differential treatment of inputs. For example, a unit might have two sites: one for inputs from Area A, the other for inputs from Area B. The unit function could specify a positive potential if and only if it is receiving input at each site. To achieve this behavior without sites, it would be necessary to use more than one unit since the links themselves bear no indication of their origin. Figure 2 illustrates the model described.

3. The Simulator

The major features of the simulator's user interface will be described in this section. It is still being developed, so changes will no doubt occur in the near future. A more detailed description with example networks will appear in the forthcoming user manual. Figure 3 shows the basic configuration of the Butterfly simulator. There is a single control program which interacts with the user. There is a sim program which runs simultaneously on several processors. The sim programs manipulate the network; the control program interprets user commands and runs the sims.

3.1. Philosophy

The simulator is meant to be as general purpose and flexible as possible. There is no attempt to force any particular model on the user. However, decisions had to be made; the kinds of networks which can be simulated are essentially those described in the previous section.

The behavior of the units is defined by functions written by the user. The functions are written in C and loaded with the simulator. They are passed the structures defining the units and links and make whatever changes are desired. This does not mean that we accept a definition of connectionist models which allows all the manipulations possible.

3.2. Integer arithmetic

Because the Butterfly does not have floating point hardware, we have to decided to represent all values as integers (shorts, actually). In order to provide more precision, users can expand the interval used in their model. For example, instead of using potentials in the range [-10,10] they can use integers between -1000 and 1000. Weights are more difficult to deal with as multiplying by 0.25 cannot be mapped to an integer multiply. One possible convention would be to consider the weight as a fraction of 100. The site function would first multiply the input by the weight and then divide by 100. These manipulations are unpleasant, but provide about an order of magnitude speedup.

3.3. Network Representation

3.3.1. Units

The basic data structure is an array of units. The fundamental identification of a unit is by its index in the array. The units are represented by a C structure:

```
typedef short Output;
typedef struct unit{
    char * type;
    /* these two fields are for internal use */
    /* they point to this unit's name table entry */
    short name_tab;
    short name_offset;
    int (* unit_f)();          /* function pointer */
    short init_potential;
    short potential;
    short rest_potential;
    Output output;
    short init_state;
    short state;
    unsigned int sets;          /* set membership bits */
    unsigned int ubits;         /* general purpose bits */
    Site * sites;               /* array of sites */
} Unit;
```

The *type* is just a character string. It serves no function beyond user identification. *Unit_f* is a pointer to the user-supplied unit function. How it is linked will be described in section 4.10. *Init_potential* is the potential the unit is given when the network is reset. *Potential* is the unit's current potential. *Rest_potential* represents the potential of the unit when at rest. It can be used, for example, to dynamically change the threshold of a unit. *Output* is the output of

the unit.

Init_state is the state the unit is given when the network is reset. *State* is the unit's current state. The states are shorts instead of character strings for efficiency. The user can define a mapping between state numbers and symbolic descriptions to make output more meaningful. The same mapping can be represented by macro definitions in the unit function, e.g.

```
#define EXHAUSTED 1
.
.
if(unit->state == EXHAUSTED){ ... }
```

Sets is a bit vector representation of the sets to which this unit belongs. Sets are a recent addition to the simulator. They are not intended to influence the behavior of the networks during a simulation, but to give the user more control over displaying and modifying units. However, they may also be useful for debugging networks. Since the user code has access to the set vector, it could do such things as ignore units in certain sets or freeze the execution there. Exactly how useful sets prove to be will be determined in practice.

Ubits is a vector of general purpose bits. One is used as a show bit, which marks the unit as one to display when the user does a show. One is used as a change bit: it is set by the simulator if the potential of the unit changes during a step of the simulation. The other bits have not yet been fixed. *Sites* is a linked list of site structures.

3.3.2. Sites

There must be at least one site if the unit is to receive inputs. The sites are represented by the following data structure:

```
/* site */
typedef struct site{
    char * name;
    int (* site_f)();
    short value;
    Link * inputs;      /* array of inputs */
    struct site *next;
} Site;
```

Name is a string which is the name of the site. *Site_f* is a pointer to the site function, which processes the inputs to that site. It assigns a *value* to the site. The unit function typically looks only at this value and not at the inputs directly. Of course, the user function can get at the inputs if desired. *Inputs* is a linked list of input structures representing the inputs to the site.

3.3.3. Inputs

The inputs are represented by the following structure:

```
/* link */
typedef struct link{
   int (* link_f)();
   short weight;
   short hist;
   Output * value;
   int from_unit;
   struct link *next;
} Link;
```

The weight and history of the link are stored in *weight* and *history*. *Link_f* is called each step of the simulation. It modifies the weight and history, if desired. The ouput of the unit at the sending end of the link is pointed to by *value*. For the user functions, (*value * weight) is the weighted input. The index of the sending unit is stored in *from_unit* for displaying. This allows backward tracing of links (forward tracing is all but impossible).

3.4. Building Networks

3.4.1. Sequential Building

Because of the large size of the networks which can fit on the Butterfly, we are most interested in building the networks each time they are run, though we will probably add the capability to load networks from UNIX files (or from Chrysalis files when we get them). This will be necessary to preserve changes which occur in networks which learn.

Networks are built in the sequential simulator by user programs written in C. The same routines will work with the Butterfly simulator. They must be loaded with the control program. They build the network sequentially across the switch. The user interactively invokes the routine with the *call* command. The following functions are provided by the simulator to facilitate building networks:

```
int MakeUnit(type,func,ipot,pot,rpot,out,istate,state,sets,ubits)
   char *type,*func;
   int ipot,pot,rpot,out,istate,state;
   unsigned int sets,ubits;

AddSite(unit,name,func)
   int unit;
   char *name,*func;

MakeLink(from,to,site,weight,history,func)
   int from,to,weight,history;
   char *func,*site;
```

Here is a sample program which builds a network with random connections.

```
#include <chrys.h>
#include <stdio.h>
#include "control.h" /* this program is run by the control process */

build()
{
   int u,i,j,unc,lc;
   char buf[20];
   long random();

   printf("\nenter #units #links each: ");
   scanf("%d %d",&unc,&lc);

   /* make units */
   for(i = 0;i < unc;i++) {
      u = MakeUnit("neuron","unit_func",0,1,2,3,0,1,0,0);
      AddSite(u,"site1","site_func");
   }

   /* make links */
   for(i = 0;i < unc;i++)
      for(j = 1; j <= lc;j++)
         MakeLink(random()%unc,i,"site1",1,0,"link_func");

}
```

Each unit has only one site named "site1". Notice that there is no notion of which processor the unit is going to be created on. The sims are filled up in order. The index returned by MakeUnit is the global index of the unit. This index is used to identify the unit in calls to MakeSite and MakeLink.

3.4.2. Parallel Building

Because of the large size of the networks, building them sequentially can take three or more hours (see section 5.2). In order to allow the user to build networks in parallel, I have added an rcall command -- "rcall all rbuild 30" causes the user function rbuild to be called for each sim with the argument 30. Rbuild must be loaded with the sim program. The sims know how many other sims there are and how many units have been allocated for each (they are all the same), so simple networks can be built in parallel. This build runs in a different environment and the functions have different implementations than the build program which is loaded with the control program. It can only make local units. The indices used are local. The one exception to this is the source unit of the link, which is a global index.

```
#include <chrys.h>
#include <stdio.h>
#include "sim.h" /* this program is run by the sim process */
rbuild(lc)
   int lc;
{
```

```
    int u,i,j,unc;
    char buf[20];
    long random();

    srandom(MySimNumber); /* initialize random number generator */
    unc = NumberOfSims * UnitsEach; /* global unit count */

    /* make all local units */
    for(i = 0;i < UnitsEach;i++) {
        u = MakeUnit("neuron","unit_func",0,1,2,3,0,1,0,0);
        AddSite(u,"site1","site_func");
    }

    /* for each local unit, make lc random links */
    for(i = 0;i < UnitsEach;i++){
        for(j = 1; j <= lc;j++)
            MakeLink(random()%unc,i,"site1",1,2,"link_func");
    }
}
```

Because all the activity is local, building networks in parallel can be faster by more than a factor of 100 when 100 processors are used.

More complicated networks may prove difficult to build in parallel. It may be easier to build the units sequentially and then make the links in parallel. A substantial time savings would still be realized since most of the build time is used for links.

3.4.3. Naming units

Units may be named individually. In addition, a consecutive block of units may be given a single name and treated as a vector. The function NameUnits declares a name. The name goes into a global name table and may be accessed from any sim or the control program.

```
NameUnit(name,type,index,length)
    char *name;
    int type,index,length;
```

Type is either SCALAR or VECTOR. Index is the global index of the unit (or first unit of the vector). Length is for vectors. When a member of a vector is displayed, the name is given with the index in brackets, e.g. "level1[39]". The user interface recognizes names in this format as well.

The names are stored in a distributed hash table. Lookup is fast enough for build programs to efficiently use it when making links. Eventually, we would like to support more sophisticated unit data structures.

3.5. Simulating Networks

3.5.1. Start up

The user must provide a build function and the unit, site, and link functions. In addition, a function which provides the mapping from function names to pointers is necessary (described in section 4.10). We have a utility which will automatically build one from a list of user functions.

These functions must be loaded with the simulator. There are two programs used by the Butterfly simulator: a control program and a sim program. A template must be made for each. A sequential build function must be loaded into the control template. The other functions are loaded into the sim template. There are shell scripts to do this, so the user types something like

```
makebsim random.funs.o68  /u/bbin
makebcontrol build.o68 /u/bbin
```

The first argument is a file to be loaded: the second is the directory in which to put the executable .68 files.

To run the simulator, the user types "control" to bshell to start the control process. It will ask how many processors to use (there will be one sim process each) and how many units to allocate on each. Then it starts the sim processes. When internal startup communication is done, the control process interactively prompts for commands and execute them.

3.5.2. Command Interface

Many commands expect a unit identifier (<uid>). This can be either a global index, a unit name, a range of units, e.g. bottom[3] - bottom[100], or a set name. Some of the commands available are:

reset - resets the network
go [<count>] - does count steps of simulation
list units - lists key information about each unit
display unit <uid> - gives complete information about the specified unit(s).
show on | off - turn on or off the show facility (explained below)
show pot <num> - when showing, display all units with potential greater than <num>
pot <uid> <num> - set the potential of the specified unit(s) to <num>
out <uid> <num> - set the output of the specified unit(s) to <num>
state <uid> <state> - set the state of the specified unit(s) to <state>
call <func name> [<num>] - call the named function, which must have been loaded with the control program. If num is present, it is passed to the function, otherwise 0 is passed.
rcall <which> <func name> [<num>] - call the named function at the specified sim. If <which> is "all", call it at all sims.
pipe <on> | <off> - determines whether output from show, list and display are piped or just written to the terminal. See section 3.5.5.
pipe "<command>" - which command to pipe the output through if pipe is on.
read <file> - read commands from the file.

3.5.3. Execution

When the go command is given, the units are simulated in sequence. The outputs they see are from the previous step, so the order of simulation is not important. Each sim does the following.

```
for each (local) unit, u
  for each site of u, s
    call the site function, passing it the site structure
    for each input to s
      call the link function,
      passing it the link structure and the unit structure
  call the unit function, passing it the unit structure
```

Notice that the link function is passed the unit structure. This is because the weight change may depend on the state or potential of the unit. When each sim has finished, they write their new outputs into the output arrays used for communication.

3.5.4. Showing

If show is turned on, the units which have their show flag on or which have a potential greater than the show potential are displayed after each step (or every *n* steps) of the simulation. This has been adequate for small networks, but we are looking into more sophisticated control over display. Note that the user functions can set the show bit. It could be set, for example, so that only just-changed units of a certain type are shown. And the user functions can themselves display information.

3.5.5. Piping output to UNIX

The standard interface for Butterfly users is a Sun workstation, which has the capability for several windows, each running a UNIX shell. Using a local package called pipsys, it is possible to start a program in a window which will wait to receive popen and system calls from the Butterfly. When the simulator pipes its output, the control process opens a pipe via the waiting program to the specified UNIX command. The default is "more".

3.5.6. Reading UNIX files

Popen can also be used for reading from a pipe. In order to use the read command, the pipsys program must be running. "Read setup" does a popen("cat setup","r"). The contents of the file are treated as user commands. At the end-of-file, commands are once again read from the terminal. Read is useful for involved or often-repeated setups, running demonstrations, etc.

3.5.7. Sophisticated manipulation of networks

The command interface is currently fairly primitive. The only modification commands which exist allow the user to change the potential, state and output of units as well as reset the network. Structural changes are not possible, except

through further build calls in user code. With smaller networks, changing the build function, recompiling and rebuilding takes only a few minutes. With the huge networks possible on the Butterfly it could take hours. Debugging code on smaller networks when possible will help, as would more sophisticated dynamic modification.

Some networks may benefit from a specialized user interface. There are functions corresponding to the interactive commands which can be called from user code. For example, the input for a parsing network may be an English sentence. A user function could map the input words to the corresponding units with a name lookup, set the potentials, and run the network the required number of steps.

4. The Implementation

4.1. Overview of the Butterfly

This section will give a brief overview of the architecture and operating system of the Butterfly. Our Butterfly has 120 processor nodes, each consisting of an 8 Megahertz Motorola 68000, 1 Mbyte of memory, and some switch hardware. The processor nodes are connected by a high speed switch consisting of 4x4 crossbar nodes laid out as a radix four Fast Fourier Transform or "butterfly" network.

The hardware allows mappings between segments in a process's address space and memory on a remote processor. Each process is limited to at most 256 of the available hardware Segment Address Registers (SAR), which means that at most 256 remote memory blocks may be mapped in to its address space. Each block is no more than 64 Kbytes, so only a fraction of the remote memory can be mapped in at any one time.

If a process wants to share memory, it must first make a memory object with the Chrysalis Make_Obj call. This returns an Object Identifier (OID) which provides a unique, global identification of that object. Any other process with that OID may map that memory into its address space (if the protection allows it).

Besides shared memory, the simulator uses events to communicate. A process which creates an event receives an Event Handle (EH) which it may share. Other process with the EH may post the event, which will be detected by the owner whenever it checks on that event (there are no asynchronous interrupts). A posted event contains one word of data. There is also a more flexible message passing mechanism called a dual queue, but this is not used directly by the simulator.

A memory reference which must go across the switch is slower than a local memory reference by a factor of two for writes and a factor of six for reads. Thus, if remote memory is going to be accessed often, it pays to make a local copy. There is an efficient mechanism for moving blocks of contiguous data across the switch called a block transfer.

4.2. Overview of the simulator

The control process typically runs on the king node. The n sim processes run one per node. They are numbered 0 through n-1 (see Figure 3). Each is passed its number and the number of sims in the command line.

4.3. Network Representation

The unit array is cut into equal length pieces, one per sim, so it is easy to map a global index to a sim number and offset. Each sim has a corresponding output array used for communication. Each sim maps in the output arrays of the other sims (see Figure 4). There is a local variable OutputArrays which is an array of arrays of shorts. OutputArrays[i] is the output array of sim i mapped in as described below. Code which accesses OutputArrays[0][3] is going across the switch to the node sim 0 is running on (unless it is already there) and retrieving the output of the fourth unit there.

Site structures are stored as a linked list attached to a unit structure and input structures are stored as a linked list attached to a site structure. Linked lists make dynamic creation easy.

4.4. Communication between control and sims

Control handles the user interface. It reads and parses the commands and sends them to the sims when appropriate. The communication scheme used is simple but quick. Control allocates a command structure which has enough fields to represent the data of any command. This is mapped in by each sim. Since they share one command description, it is not possible for two sims to be executing two different commands at the same time. There seems to be no serious memory contention for this one structure. Commands which execute in parallel have, as it happens, little data, and it is read only once. For make commands, enough data must be sent to justify a block transfer of the command structure to a local copy.

Each sim creates a command event and gives it to control. When control wants to send a command, it posts the event(s) and waits for the sims to finish. When a sim receives a command event, it executes the indicated command and increments a global short variable. Control busy waits on the global variable. Here is the routine send_all, which sends the same command to all sims. The global command structure has already been filled in. Notice that the event data is the command code. This is enough for many commands; the command structure will not be consulted.

```
/* in control */
sendall()
{
   int who;
```

```
    /* tell all sims to execute the command */
    for(who = 0;who < no_sims;who++){
        Post_Event(simevents[who],(long)cmd->cmd);
    }

    /* busy wait for them to finish */
    while(*icount != no_sims){
          /* check to see if any have died */
        if(MReceive_Event(deathE,NULL) != NULL){
            fprintf(stderr,"termination due to death of child sim\n");
            exit(1);
        }
        Sleep(10); /* do not want to monopolize icount */
    }
    *icount = 0;
}
```

There is a similar function send_one, which sends a command to a single sim. It is used when the command involves a unit on one sim or to send commands synchronously: e.g. a list command must finish on sim 2 before beginning on sim 3 or the output will be mixed. Here is the code from the sims which waits for the above event.

```
/* in sim */
while(1){
    e = MWait(CommandE,(long)0);
    cmd = (int)Event_Data(e);
    switch (cmd) {
          .
          .
      case SHOWPOT_C :
        show_pot = Cmd->idata;          /* Cmd->idata set by control */
        break;
          .
          .
    }
    Reset_Event(CommandE); /* so it can be reused */
    Atomic_add(Icount,1);  /* let control know you are done */
} /* while(1) */
```

4.5. Start up

4.5.1. Stage one

There are three stages to the start up procedure. The control process is started by the user. It prompts for the number of sims to use and how many units each is to allocate. It then makes and declares global names for several objects the sims will lookup and map in. It must make an array of Event Handles. Each sim will place an event in its slot which control will use to inform it there is a command.

```
simeventsOID = Make_Obj(0,-1,sizeof(EH) * no_sims,RW_rw_);
simevents = (EH *) Map_Obj(simeventsOID,0,RW_rw_);
Name_Bind("Elist",simeventsOID, NTYPE_OBJ); /* Chrysalis name table */
```

Similarly, it makes and names a command structure. Each sim will map this in and refer to it for command arguments when necessary. Control also makes and declares an object large enough to hold no_sims OIDs. This will be mapped in by the sims. Each will write the OID for its ouput array here. Control calls a name table initialization routine, which will be described in section 4.7. The last global object control makes is the short variable Icount. This is incremented by the sims when they finish a command.

Control now starts all the sims. This marks the end of stage one. The call to LoadStart in the following code starts the named program, downloading it if necessary. There are some special considerations which should be pointed out. First, the sims are started as child processes so that they will all be stopped if control dies. Because each sim maps in the output array of every other sim and because of the number of mallocs they do when building a network, they need a large number of SARs. LoadStart asks for 150 (which actually grabs 256). Also, because of the large number of mallocs done, cp_msegsize is set to the maximum of 64000. This could result in out-of-memory errors when there are actually blocks smaller than 64000 left (we may fix this in the future).

```
/* in control */

deathE = Make_Event(0,0,0,0); /* event is posted upon death of child */

/* start all sims */
for(i = 0;i < no_sims;i++){
   char *argv[5],buf1[20],buf2[20],buf3[20]; /* to setup command line */

   argv[0] = "sim";
   sprintf(buf1,"%d",i);          /* this sim's number */
   argv[1] = buf1;
   sprintf(buf2,"%d",no_sims);    /* number of sims */
   argv[2] = buf2;
   sprintf(buf3,"%d",units_each); /* number of units to allocate */
   argv[3] = buf3;
   LoadStart("sim",nextprocessor(),4,argv,deathE);
}
```

```
/* wait for them all to finish stage two */
while(*icount != no_sims) {
   if(MReceive_Event(deathE,NULL) != NULL){
        fprintf(stderr,"termination due to death of child\n");
        exit(1);
   }
   Sleep(10);
}
*icount = 0;
```

4.5.2. Stage two

Stage two of the startup is carried out by the sims while control busy waits. Each sim interprets its command line arguments. They find and map in the global output array OID table and enter their own output array OID.

```
/* in sims */
OlistOID = Find_Value("Olist",NTYPE_OBJ);
Olist = (OID *)Map_Obj(OlistOID,0,RW_rw_);
MyOutputArrayOID = Make_Obj(0, -1, last_unit*sizeof(Output), RW_rw_);
Olist[MySimNumber] = MyOutputArrayOID;
/* outputs is the local name for this sim's output array */
outputs = (Output *) Map_Obj(MyOutputArrayOID,0,RW_rw_);
```

Each sim maps in control's event array, enters an event, and unmaps the array. Each sim maps in the command structure and calls its name table initialization routine (section 4.7). Finally, each sim maps in the variable Icount. They then increment Icount, do some local initialization, and enter a command loop. This is the end of stage two.

4.5.3. Stage three

When the sims finish stage two, control sends each a mapin command which causes them to map in the output arrays of every other sim.

```
/* in sims */
for (i=0; i<NumberOfSims; i++)
  if (i != MySimNumber) /* my output array already mapped in */
        OutputArrays[i] = (Output *) Map_Obj(Olist[i],0,RW_rw_);
  else OutputArrays[i] = outputs;
```

When this commands completes, control enters the command loop where it prompts for, reads and executes user commands.

4.6. Building networks

4.6.1. Sequential building

Build programs which run with control send make commands across the switch. The fields in the global command structure are filled in with the relevant values for the unit, site or link and the command event for the appropriate sim is

posted. Control knows how many units have already been made (by it), so it knows where the next one goes. It sends its idea of which local unit structure should be used for the unit being made. If it is not consistent with the sim an error is generated.

The sim command interpreter calls the local make functions with the appropriate data from the command structure. When a unit is being made, the structure already exists and is just filled in. When a site or link is being added, the space must be allocated. In order to avoid doing a malloc for each site and link made (to save time and space), space for several is malloced when there is a need and individual structures are assigned from this.

4.6.2. Concurrent building

When networks are built concurrently, control loses track of what is where. The sims are each in charge of their own piece of the unit array. The local make functions are called directly. They can, perhaps, orient themselves by doing name lookups.

4.7. Global name table

Each sim allocates a piece of the name table. The table is an array of name structures:

```
typedef struct n_i_desc {
        char    name[22];       /* Pointer to name */
        short   type;           /* Type of unit {0, 1} */
        int     index,          /* Index of first unit */
                size;           /* Size of vector if type 1 */
} NameDesc;
```

Each local table holds 2039 records, the largest prime number of records which will fit in 64 Kbytes of memory. When the entire Butterfly is being used, there will be over 240,000 records all together, which should be enough even if every unit has a distinct name.

Much as with the output arrays, control makes and names a table called Tables during start up which will hold the OIDs for each local name table. Every sim maps this table in, and enters the OID of its name table. The name tables are not mapped in by each sim because of SAR scarcity, but each sim has the OID for each table.

Each sim also allocates a short and puts the OID for it into another global table called Locks. Locking is used when inserting names into a table to prevent two processes from finding and using the same record. Locking is not necessary when looking up names.

When a name is declared, the location in the table is stored with all units which have been named. This allows efficient mapping from unit index to name.

The following is the hash function used:

```
n = name;
for(hash = 0,shift = 0; *n != ' ';shift = (shift+3)%11,n++)
   hash += ((*n) << shift) + *n;
tab = hash % no_tables;
offset = hash % NO_RECORDS;
```

This function has a good distribution even when there are many names which differ by only one or two characters. The one hash value is taken modulo the number of tables to find the local table to be used, and modulo the number of records per table to find the offset within the table. In the event of collisions, a linear search from the point of collision is made. Following is the code for insertion as implemented for control. The sim code is the same except for the last three lines. Control must send a command to enter the name table index for each unit named. The sim code directly modifies the unit structure (note that this means an off-node unit cannot be named by sim code).

```
/* map in lock variable and busy wait for that table to be free */
lock = (short *) Map_Obj(lock_list[tab],0,RW_rw_);
while(Atomic_add(lock,1)){
   Atomic_add(lock,-1);
   Sleep(1);
}
/* map in table */
table = (NameDesc *) Map_Obj(table_list[tab],0,RW_rw_);

/* search for first free record starting at offset */
for(i = offset,ct = 0;table[i].size && strcmp(table[i].name,name) &&
   ct <= NO_RECORDS;ct++, i = (i+1)%NO_RECORDS);
if(ct > NO_RECORDS){
   fprintf(stderr,"name table %d full; did not insert %s\n",tab,name);
   RETURN(-1);
}
if(table[i].size){
    fprintf(stderr,"name already in table: %s\n",name);
}
else {
   table[i].type = type;
   table[i].index = ABSOLUTE_IND(index); /* passed local index */
   table[i].size = length;
   if(strlen(name) > 21) name[21] = '\0';
   Do_bt(name,table[i].name,strlen(name)+1);
}
Atomic_add(lock,-1);              /* unlock table */
Unmap_Obj(table_list[tab],table);         /* unmap */
Unmap_Obj(lock_list[tab],lock);
/* pack table and offset into one int to send to sim */
namecode = ((tab << 16) | i);
for(s = index;s < index + length;s++)
   /* Send_Cmd converts index s into sim number */
   Send_Cmd(MOD_NAME,s,namecode,"");
```

In order to test a record to see if it already contains the name, a strcmp is done across the switch. In another implementation, the remote string is block transferred and the strcmp is done locally. In tests of the name server, the two implementations took about the same amount of time. The block transfer is more efficient per byte, but always retrieves 22 bytes; the strcmp retrieves only as many as necessary. To copy the string into the table record, a block transfer is used. This should be more efficient than copying each character individually.

The code for finding a name's entry is similar, except the table need not be locked. A block transfer is used to copy the remote record to a local record.

4.8. Simulating

When the user types a go command, the following happens.

```
clock + = 1;
/* tell each sim to do one step */
Send_Cmd(GO_C,EVERY,0,"");
/* when they are done, tell them to update ouput arrays */
Send_Cmd(UPDATE_C,EVERY,0,"");
if((i % echo_step) = = 0) printf("End of step %d\n",clock);
if(show && (i % show_step) = = 0){
   if(pipeflag)
        Send_Pipe_Cmd(SHOW_C,EVERY,0,"",pipecommand);
   else
        Send_Sync_Cmd(SHOW_C,EVERY,0,"");
   }
}
```

The sims must wait until everyone is done before updating their output arrays. A show is a listing of certain units as described in section 3.5.4. Send_Pipe_Cmd is described in the next section. Send_Sync_Cmd waits for the current sim to finish before sending the command to the next. This keeps the output in order.

4.9. Piping output to UNIX

Before executing a popen, the user must start the pipsys program in some UNIX shell. It responds with its network port number. On the Butterfly, the network code must be downloaded and the Chrysalis environment variable PIPSYS set to this number. Any program loaded with the pipsys library can now do a popen.

Rather than have each sim open a pipe, which would result in discontinuous output, control opens the pipe and the sims send their output to control using a stream. Here is how Send_Pipe_Cmd works.

```
cmd->pipeflag = 1;
cmd->pipeQ = ComQ;          /* ComQ is a dual queue handle */
```

```
catch
   pf = popen(pipec,"w");        /* pipec is UNIX command to pipe to */
onthrow
   when(TRUE){
        fprintf(stderr,"cannot open pipe: %s\n",throwtext);
        return 0;
   }
endcatch;

for(i = 0;i < no_sims;i++){
   cf = stream_fdopen(0,ComQ,"r");     /* 0 meaningless */
   if(cf == NULL){
        fprintf(stderr,"control could not open pipe stream\n");
        RETURN(0);
   }
   Post_Event(simevents[i],(long)cmd->cmd);/*tells sim to look at cmd*/
   /* read from stream */
   while(1){
        readct = read(cf,readbuff,BLEN);
        if(readct == 0){          /* output done */
           close(cf);      /* close stream */
           break;
        }
        else write(pf,readbuff,readct);     /* write to pipe */
   }
   while(!*icount)
        if(MReceive_Event(deathE,NULL) != NULL){
           fprintf(stderr,"termination due to death of slave\n");
           RETURN(0);
        }
   *icount = 0;
}
close(pf);                    /* close pipe */
```

The fact that the output is to be piped is communicated to the sims by the pipeflag field of the command structure. The dual queue required for the stream is also put there.

The catch around the popen makes the program more robust. If the pipe cannot be opened, a throw will be generated which would kill the program if not caught.

The sims surround all output which might be piped with pipe_begin() and pipe_end() calls. All output is written to the file dispf.

```
/* in sims */
pipe_begin()
{
   if(Cmd->pipeflag) /* output is to be piped */
      if((dispf = stream_fdopen(5,Cmd->pipeQ,"w")) != NULL){
         return;
      }
```

```
        else{
            fprintf(stderr,"#%d sim: cannot open stream0.MySimNumber);
            exit(1);
        }
    else dispf = stdout;
}

pipe_end()
{
    fflush(dispf);
    if(Cmd->pipeflag && dispf != stdout){
        close(dispf);
    }
}
```

4.10. Mapping function names to pointers

The make functions need to find a function pointer given its name. So do call and rcall. In order to do this, a table is built by the user-provided function bind_func. The table is also be used to map function pointers to names for display. This is why function names need not be stored in the unit, site and link structures. The table building function looks like:

```
#include "/u/connect/src/sim/names.h"

char *store_string();
int link_func();
int build();
FuncTable *bind_func(length)
    int *length;
{
    int i;
    FuncTable *table;

    table = (FuncTable *) malloc(sizeof(FuncTable) * 2);
    *length = SIZE;
    table[0].name = store_string("link_func");
    table[0].func = link_func;
    table[1].name = store_string("build");
    table[1].func = build;
    return table;
}
```

There is a utility to generate this function from a list of names, and we may generate it directly from the name table of the user's .o68 file in the future.

5. Performance

5.1. How large a network will fit

Each unit uses 40 bytes. Each site and link uses 20 bytes. Because the largest array which can be allocated is 64 Kbytes, 1599 units per sim is the maximum. This figure could be increased in a number of ways. The unit array could be split

into two or more parallel arrays, for example. This would leave less space for links, of course.

There is 1 Mbyte of memory on each processor. Chrysalis takes about 90 Kbytes, leaving 14 chunks of 64 Kbytes each. The space for sites and links is allocated using malloc. Since the malloc size is 64 Kbytes, chunks of memory smaller than this cannot be used during a build. The sim template uses about 20 Kbytes, leaving 13 64 Kbyte chunks. The name table grabs 64 Kbytes (this will probably be made adjustable) leaving 12 chunks. If there are enough units, they will use another 64 Kbyte chunk, leaving 11 for sites and links. This means the total number of sites and links must be less than 35,200 per sim. I have ignored overhead space used by malloc, so call it 35,000. This could be increased by about 10% by making the name table smaller (so it uses a smaller chunk) or by making malloc more flexible.

5.2. Build times

The tests were conducted using 100 sim processes. The code used is that in section 3.4.1. The network built had 100,000 units and 3,000,000 links, 30 links to each unit from a random source. When it was built sequentially from the control process, the units and sites took 9 minutes to build and the links took 2.75 hours. When a network of the same size was built concurrently, the total time for units and links was 56 seconds. The speedup was due to concurrency and locality. This difference makes us anxious to develop concurrent build techniques for all our networks. The speedup *would* still be dramatic even if the units were built sequentially and the links made in parallel. This would require a "call unitbuild" followed by a "rcall all linkbuild".

5.3. Name table performance

When the above tests were run no names were used. I have conducted independent tests of the name table software. 90 processes were used, one per processor. Each allocated a 64 Kbyte block of memory -- 2039 name records. So the global name table had over 180,000 name records.

Each process entered 1000 distinct names (90,000 total names) simultaneously. This took about 6 seconds. Each process then looked up all 90,000 names (8,100,000 total lookups) simultaneously. The names were generated in a different order so no two processes were looking up the same name at the same time. A lookup involved finding the record and retrieving three fields of data, one of which was generated from the name and used to check that the correct data was retrieved. The total lookup time was 409 seconds. This is fast enough so that even parallel build programs which do a name lookup for each of several million links will still run reasonably fast.

These tests involved very intense switch traffic. They encounted bus errors unless the switch timeout was increased to one second and alternate paths were enabled. Surprisingly, the intense switch traffic did not increase the running time significantly. This was determined by changing the test program so that only one

process did the 90,000 name lookups. This took 401 seconds, only 8 seconds less than when there was heavy switch traffic.

I varied the test program to leave all the local tables mapped in. This saves a Map_Obj and Unmap_Obj on each name lookup. It ran 15% faster. Since it was looking up names as fast as it could generate them, the speedup for a build program would be a much smaller percentage. In any case, there are not enough SARs to leave the name table mapped in for the sims because of the number of output tables already mapped in.

5.4. Simulation time

I ran a number of tests to measure the run time and speedup of the simulator. The same network could not be used with different numbers of processors because of limited memory (and a small network does not show much improvement on a large number of processors because the overhead of control begins to dominate the time). What I did is run each test with the same number of units per processor (1000) and the same number of links per unit (30). With perfect speedup, the simulation time would be the same no matter how many processors are used.

The user function was simple: the potential was set to the sum of the inputs. The link function set the history to the value of the input. In the first series of tests, weights were ignored: the inputs were simply summed. This represents about the fastest possible run time for a network this big. More complicated unit, site and link functions would slow it down appreciably. After all, a link function is called 30,000 times per step per sim.

For the simple network, running on one processor, the execution time was 2.0 seconds per step. Running on 90 processors, the execution time was 2.6 seconds per step. This works out to about 70 effective processors. A graph of the speed up for these and other configurations is shown in Figure 6.

As expected, when the build function was changed so that all links were from other units on the same processor, 90 sims also took 2.0 seconds per step.

In the next series of tests, the inputs were multiplied by the weights and divided by 100 (see section 3.2). This represents a more typical amount of computation. For 1 sim, the time was 4.1 seconds per step; for 90 sims the time was 4.6 seconds per step. This works out to about 81 effective processors. The speedup is better because a smaller percentage of time was spent retrieving outputs across the switch. The results of this test are given in Figure 5.

With 90 sims and 90,000 units, show was turned on with the show potential set so that only 30 units would actually be displayed. All 90,000 units had to be examined sequentially, however, to see if they were eligible for showing. This took almost 2 minutes.

I ran some tests comparing different ways of using weights. In each test the inputs to a single site were summed. The first test used no weights. The second test used integer weights and did a multiply followed by a divide as described in

section 3.2. The third test used floating point weights. The floating point code ran 6 times slower than the integer code with multiplies and divides and 20 times slower than the integer code without weights. We consider this difference sufficient to justify the somewhat troublesome manipulation of integer weights. If we acquire floating point hardware, we will probably use floating point weights and potentials, if not outputs.

6. Future work

6.1. Saving networks

There are many directions future work on the simulator may take. We will better know what we most need after more experience using it. The purpose of this section is to outline some of the changes and improvements we are considering. One of the first features we want to add is a compact format in which to save the state of the network. This is especially important for networks which learn by changing weights. The new weights cannot be recoved except by running the same simulation. A complete save would require a file tens of Megabytes big and take a long time to do. Since the topology of the network can be recreated exactly by calling build with the same parameters (and same random number seed, if appropriate), it will suffice to save the link weights. This will require less than 10 Mbytes. If we get Butterfly disks, it may be feasible to save an entire network.

6.2. Compact representations

We are considering adding the ability to represent arbitrary numbers of units with a single unit structure. Each of these units would have its own position in the output array, but they would have no other individual representation, no explicit list of inputs. The one unit function would be responsible for filling in the output array. This might be useful when modeling a retina in a vision application, for example. A 100x100 retina might have its values set once by reading from a file and not changed after that. Having a single function take care of all 10,000 units would save time and space. Such blocks of units could have inputs of a sort. The single unit structure representing them all could have a list of inputs. Sorting out which go where would be up to the user function. Of course, other units could receive inputs from these compact units since they have their output in the output array.

Another space saving technique is possible now. Instead of relying on a list of inputs, user functions could generate indices of units through name lookups or by other means and retrieve the input values by a lookup in the OutputArrays table. This would be slower and would not allow histories or weights on the links.

One more trick we have considered is the dynamic creation of units. This technique was used successfully by Sabbah (1982). It is useful when the network contains a very large number of units, but any one computation uses only a fraction of them.

6.3. Better tools

We will probably need more sophisticated network examination tools in order to deal with huge networks. We are especially interested in graphics tools to run on the Sun workstations. For example, a window which continuously displays the potentials of several hundred key units could prove very useful, as would the ability to trace the activation emanating from a single unit. We may add a list of outputs for each unit so we can trace the links either way. The reason we do not now do this is to save space, but fitting lots of units in memory does us no good if we cannot effectively work with them. We could even add a switch to turn this feature off on really large networks.

6.4. Internal improvements

Carla Ellis plans to implement her distributed hash table (Ellis, 1985) on the Butterfly. The improved name table would be extendable, allow deletions and would not impose string length limits.

I am not entirely happy with the communication mechanism between control and the sims. It does not allow different sims to be executing different commands at the same time because of the single global command structure. Consider rcalls, which could be quite involved: it might be nice to start a remote procedure on one processor and then start another somewhere else without waiting for the first to complete.

7. Discussion

The simulator is important from a systems point of view because it is an example of a complicated program which can take full advantage of large numbers of tightly-coupled processors. There are many applications for which it is difficult to divide the task into n pieces so that anything like a speedup of n occurs. The large connectionist networks cannot even run on any other machine in the department because of memory limitations, and storing networks on disk during a simulation would be too slow.

The simulator is important for connectionist researchers because it gives them a tool with which to test their theories. The behavior of connectionist networks is not always easy to predict using analysis. It is necessary to actually run them.

And connectionist networks can easily grow large. The only existing "connectionist networks" which can handle sophisticated AI tasks use billions of units (i.e. neurons in the human brain). Of course, this does not mean that this many units are necessary to do intelligent tasks, but we should not expect to get by with just a few.

Even with the Butterfly, the number of units we can simulate is five orders of magnitude less than the number of neurons in the brain. More important, the number of links is about ten orders of magnitude less. In any case, we have plenty of work to do learning how to use hundreds of thousands of units. The Butterfly connectionist simulator gives us a tool for this task.

Acknowledgments

This work would not have been possible without the help and advice of Liudvikas Bukys, Rochester's lepidopterist.

References

Ellis, C. S., "Concurrency and linear hashing", TR151, Department of Computer Science, University of Rochester, March, 1985.

Feldman, J. A. and Dana H. Ballard, "Connectionist models and their properties", *Cognitive Science*, 6, 1983, pp. 205-254.

Rumelhart, D. E. and J. L. McClelland, *Parallel Distributed Processing: Exploring the Microstructure of Cognition*, Cambridge, Mass: Bradford Books/MIT Press, 1986.

Sabbah, D., "A Connectionist Approach to Visual Recognition", TR107, Department of Computer Science, University of Rochester, April, 1982.

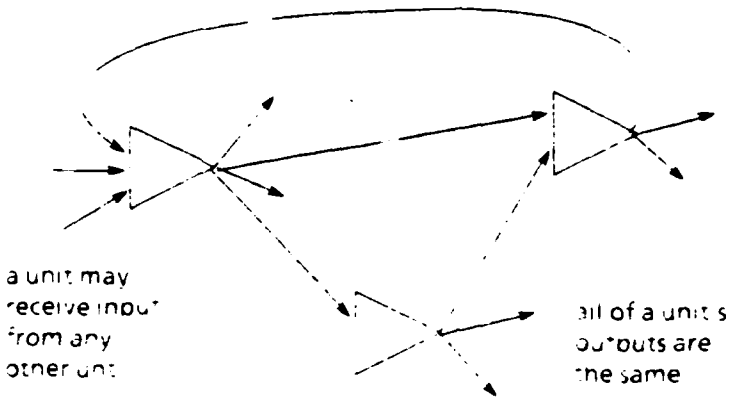


Figure 1.

Sample Units and Connections.

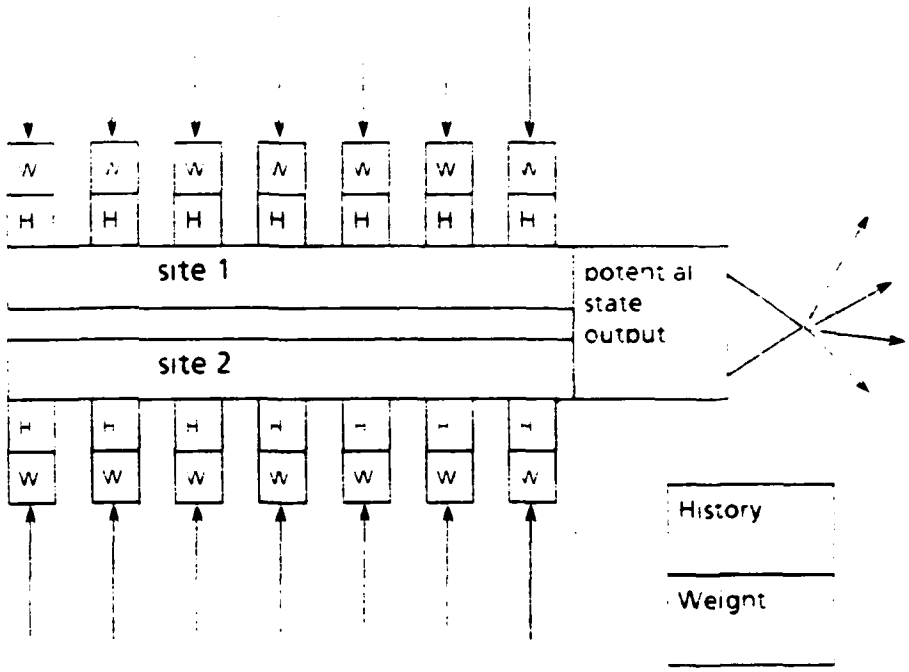


Figure 2.

Sample unit with seven connections to each of two sites.

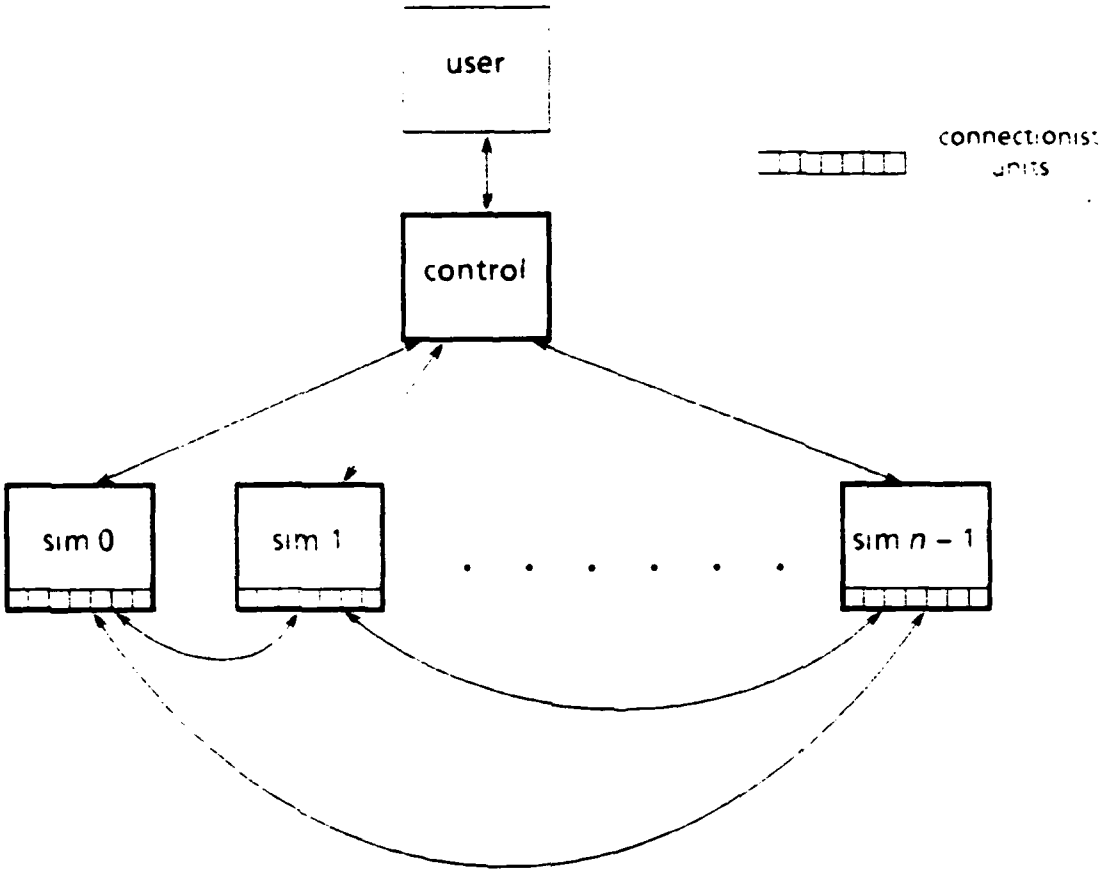


Figure 3.

Who talks to whom.

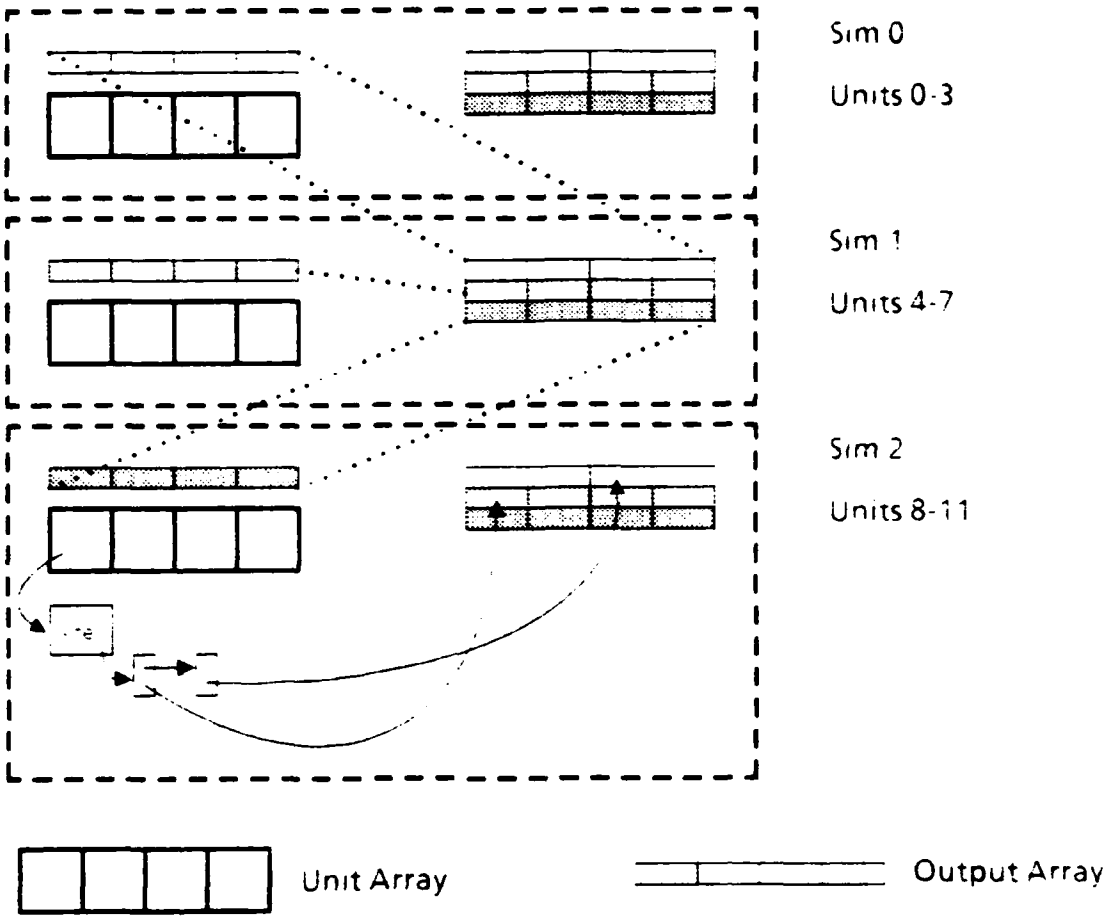


Figure 4.

Each simulator maps in the output arrays of the others. The links from unit 2 and unit 4 to unit 8 are shown

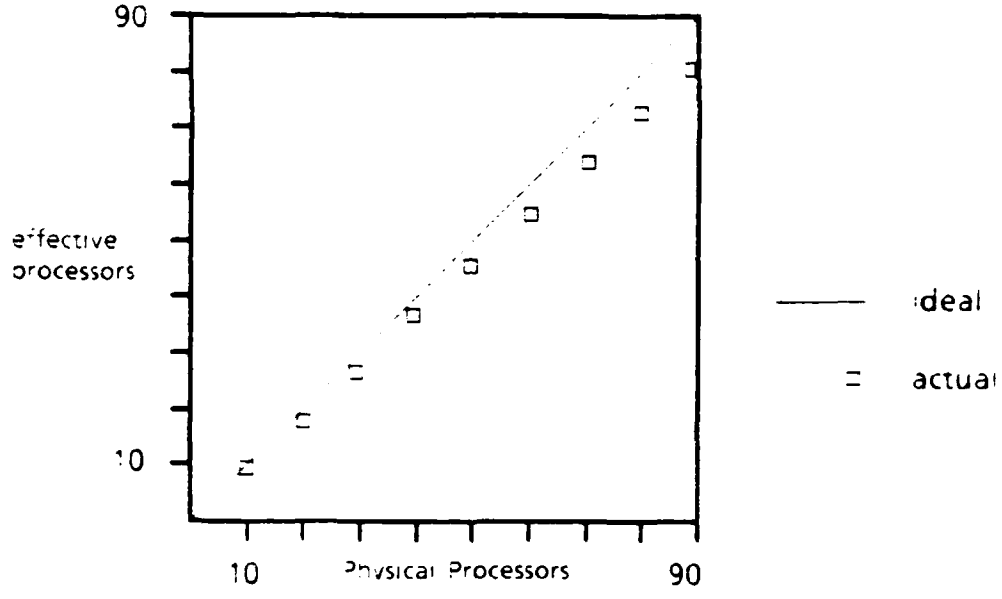


Figure 5.

Speedup under "normal" conditions

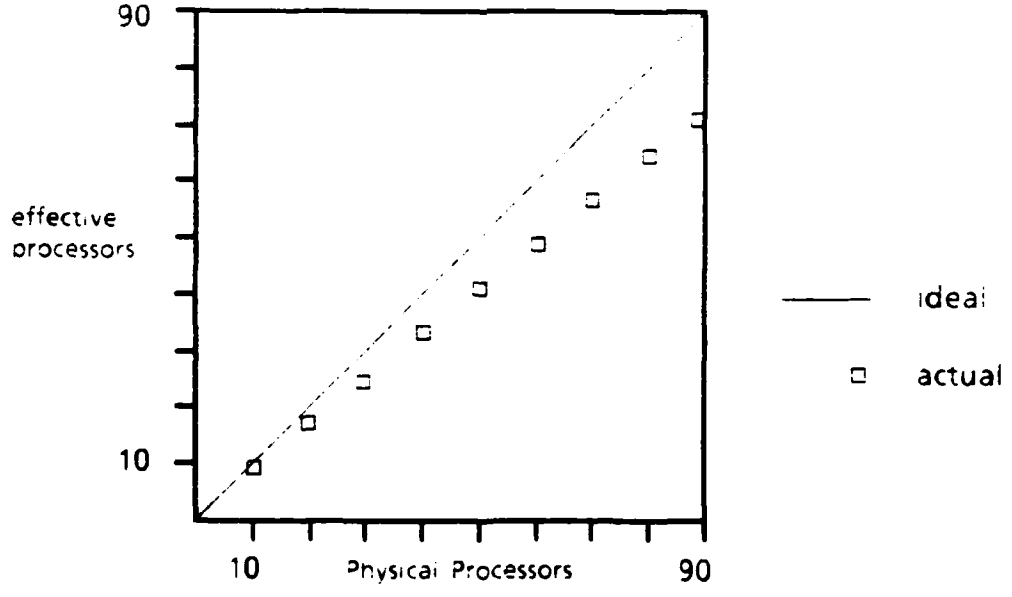


Figure 6

Worst case speedup

END

DTIC

8-86